T.C.
İSTANBUL MEDENİYET ÜNİVERSİTESİ
MÜHENDİSLİK VE DOĞA BİLİMLERİ
FAKÜLTESİ

MEZUNİYET PROJESİ

DAĞITIK GPU'LARDA HESAPLAMA

AHMET BURAK KARA – 21120101027

BERŞAN MUHAMMET EKİCİ (İSTANBUL TİCARET ÜNİVERSİTESİ)

ELEKTRİK-ELEKTRONİK MÜHENDİSLİĞİ

DANIŞMAN
Dr. Haluk Bayram

06, 2025

İSTANBUL

ÖNSÖZ

Bu bitirme tezi, TUSAŞ bünyesinde yürütülen LIFT UP programı kapsamında geliştirilen "Dağıtık GPU'larda Hesaplama" projesinin araştırma, geliştirme ve uygulama sürecini kapsamaktadır. Proje süresince yüksek hesaplama gücü gerektiren yapay zeka modellerinin eğitiminde, TUSAŞ bünyesinde mevcut olan çok sayıda GPU'nun birlikte ve verimli şekilde çalıştırılabilmesi hedeflenmiştir.

Sistem, socket tabanlı bir haberleşme protokolüyle farklı düğümlerdeki GPU'ların eşzamanlı çalışmasını sağlamaktadır. YOLOv8 modeli üzerinde yapılan uygulamada, parametre birleştirme işlemleri hem CPU hem de GPU üzerinde test edilmiş; özellikle GPU tabanlı model kombinasyonunun yüksek verim sağladığı gözlemlenmiştir. Gerçekleştirilen performans testleri ile eğitim sürelerinde ideal süreye %73'e varan oranlarda yaklaşım kaydedilmiştir. Bu sayede geliştirilen yapı, büyük ölçekli yapay zeka modellerinin daha kısa sürede eğitilmesine olanak tanımaktadır.

Projenin önemli bir bileşeni olarak geliştirilen CPU ve GPU benchmark araçları sayesinde, düğümlerin donanımsal kapasiteleri analiz edilmiş ve bu analizler doğrultusunda farklı performans seviyelerine sahip makinelerin aynı eğitim sürecine dahil olabilmesi mümkün kılınmıştır. Bu yönüyle sistem, asimetrik donanım altyapılarına sahip kümelerde de etkin çalışabilmesi öngörülmüştür.

Gerek proje süresince yönlendirme ve desteğiyle olsun gerek okulumuzda vermiş olduğu derslerde bana öğrettikleriyle katkı sağlayan akademik danışmanım Dr. Haluk Bayram'a, çalışma ortamı sağladığı için Marmara Üniversitesi'ne ve ihtiyaçlarımızı karşılamak için emek harcayan Dr. Barkın Bakır'a, projemizi 2209B programlarınca desteklemelerinden dolayı TUBİTAK' a, teknik ve mali yardımlarının yanı sıra Marmara Üniversitesi TUSAŞ Havacılık yapısalları ve Optimizasyon Laboratuvarını bize proje bağlamında sağladıklarından dolayı TUSAŞ'a ve süreç boyunca birlikte çok emek verdiğim proje arkadaşıma teşekkürü borç bilirim.

Bu çalışmanın, TUSAŞ'ın sahip olduğu bilgi işlem kaynaklarının daha etkin kullanımına katkı sunmasını temenni ederim.

Haziran 2025 Ahmet Burak Kara

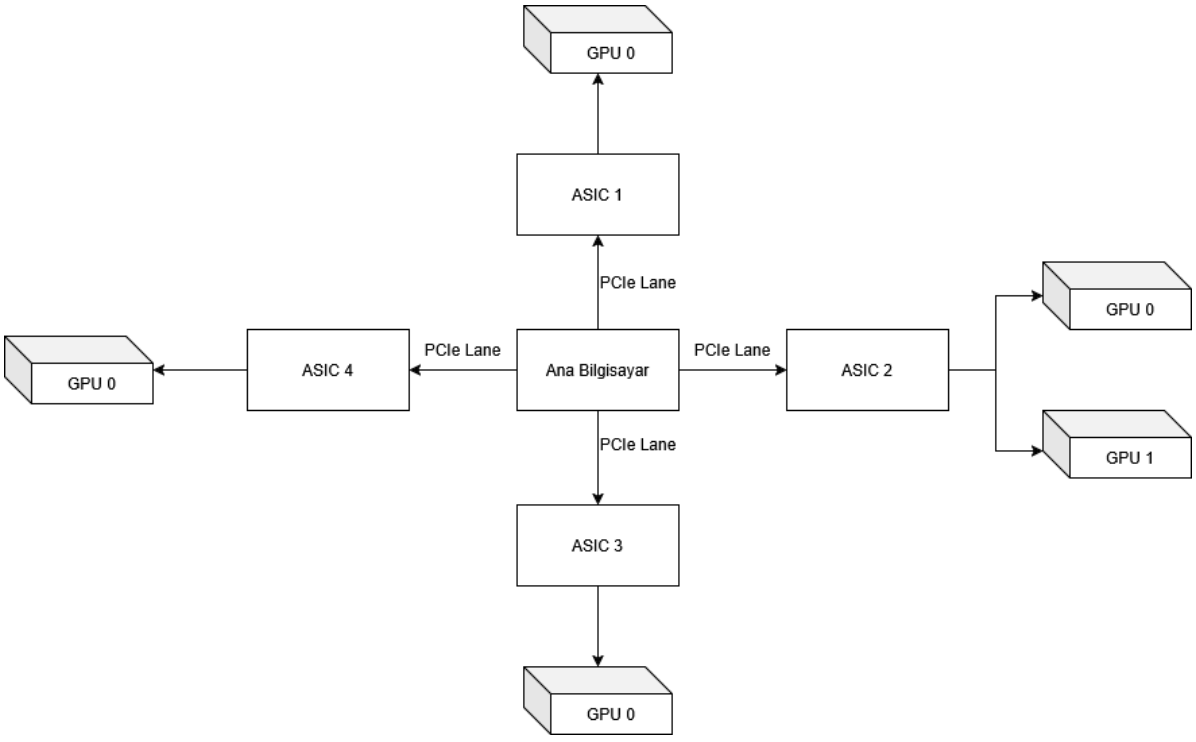
İÇİNDEKİLER

Sayfa No

1. GİRİŞ

Yapay zeka modellerinin eğitimi, özellikle bilgisayarla görü (computer vision) gibi yoğun hesaplama gerektiren alanlarda, büyük veri kümeleri üzerinde yüksek işlem gücü kullanımıyla mümkündür. Modern derin öğrenme modellerinin artan parametre boyutları ve karmaşıklığı, tek bir GPU'nun hesaplama kapasitesini hızla yetersiz hale getirmektedir. Bu durum, özellikle endüstriyel ölçekte çalışan kurumlar için eğitim süresinin uzamasına ve kaynakların verimsiz kullanılmasına yol açmaktadır.[1][2]

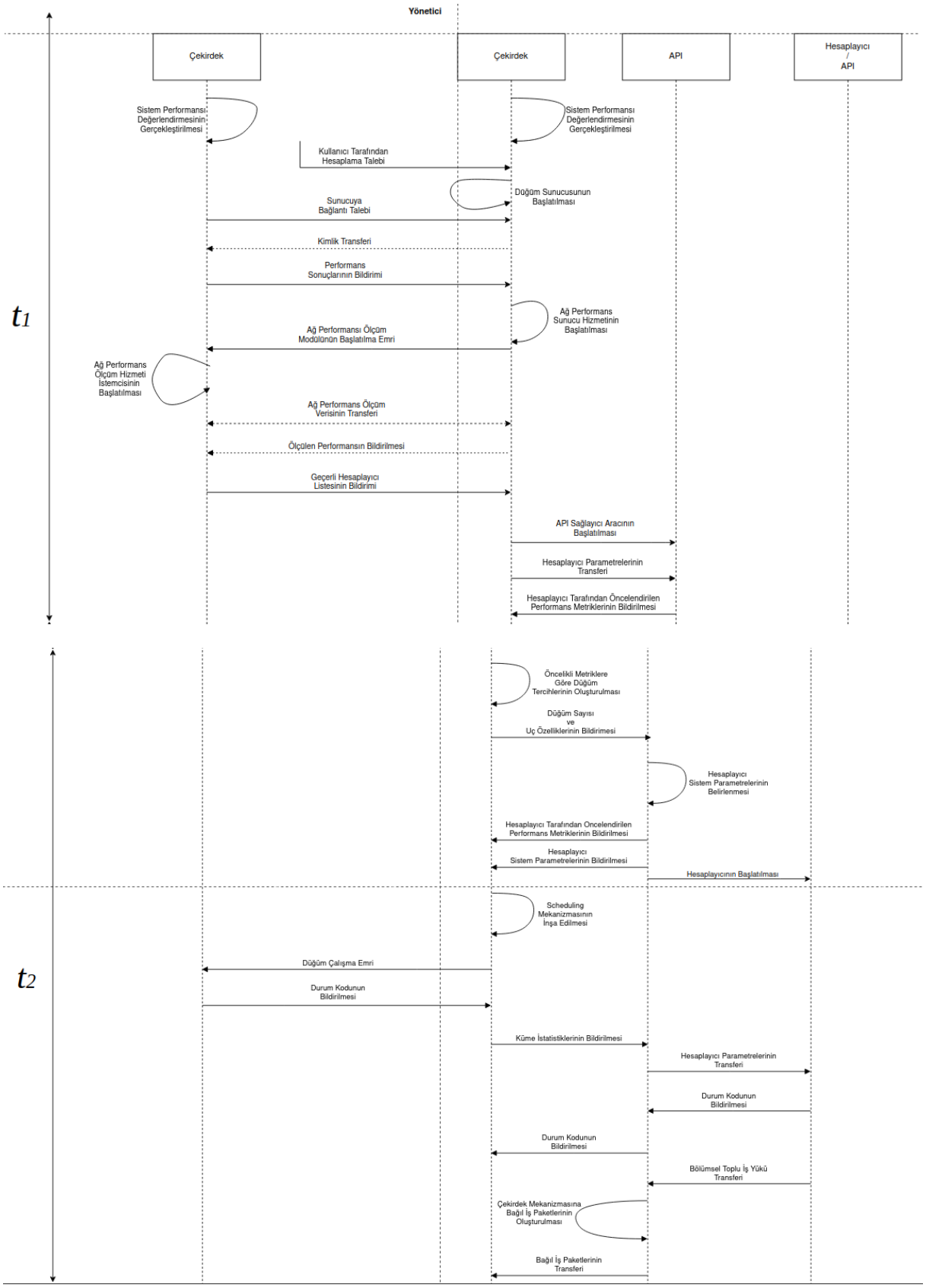
Bu sorunu aşmak için geleneksel olarak başvurulan yöntemlerden biri, çok sayıda GPU'nun merkezi bir bilgisayar üzerinde PCIe hatlarıyla bağlanarak oluşturulan süper bilgisayar yapısıdır. Bu mimaride tüm grafik işlemciler aynı donanım havuzunu paylaşır ve ortak bellek erişimi sağlar. Ancak bu sistemler yüksek kurulum maliyetine, sınırlı ölçeklenebilirliğe ve donanım esnekliği eksikliğine sahiptir. Ayrıca fiziksel olarak aynı kasa içinde GPU'ları birleştirmek, enerji tüketimi, ısı yönetimi ve donanım bakımı açısından zorluklar doğurabilmektedir.[3]

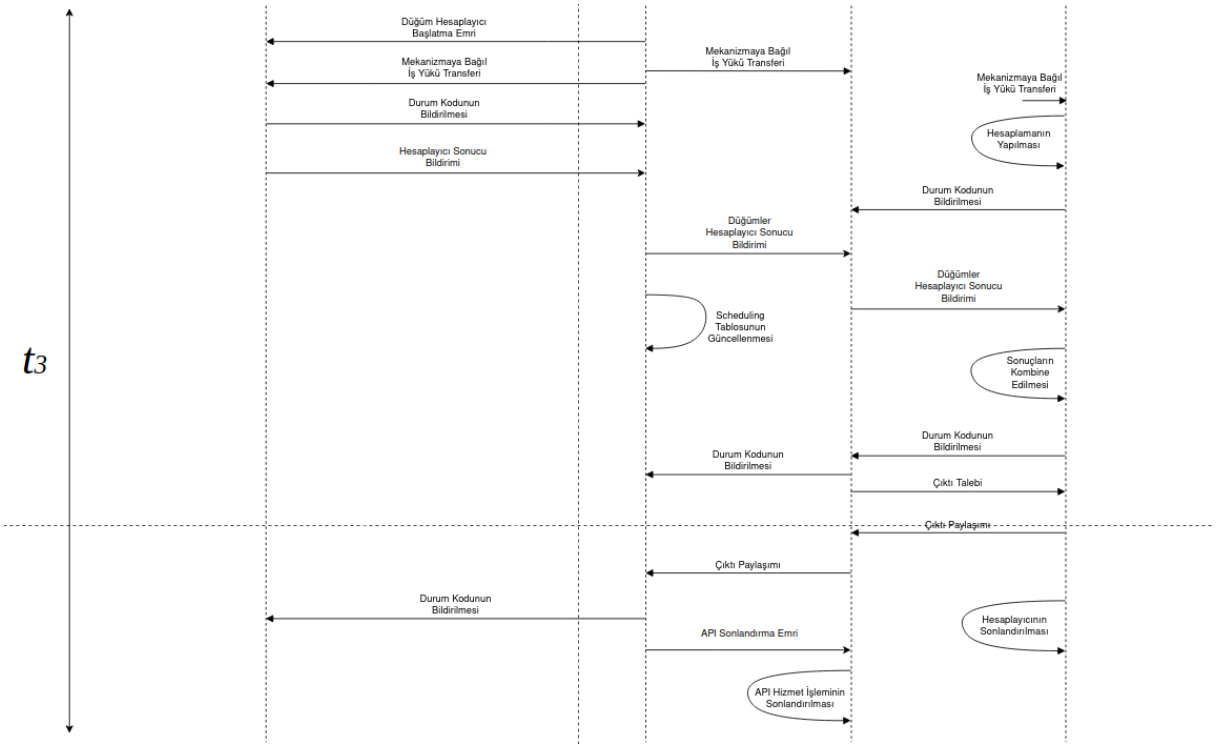


Şekil-1: ASIC ile süper bilgisayar yapısı

TUSAŞ bünyesinde bulunan çok sayıda GPU'nun potansiyelini değerlendirmek amacıyla başlatılan bu projede, süper bilgisayar yönteminin kısıtlarını aşan, ağ tabanlı dağıtık eğitim altyapısı geliştirilmiştir. Proje kapsamında, farklı fiziksel makinelerde yer alan GPU'ların bir ağ üzerinden haberleşerek tek bir modelin paralel eğitimi gerçekleştirmesi hedeflenmiştir. Bu

yaklaşım, merkezi bir süper bilgisayar yapısından farklı olarak, her makinenin kendi belleğini ve kaynaklarını koruduğu; bu sayede ölçeklenebilirliğin artırıldığı bir sistem sunar.[4]





Akış Diyagramı - 1

Bu sistem, socket tabanlı bir haberleşme protokolü kullanarak model ağırlıklarının ve kayıp fonksiyonu değerlerinin senkronize şekilde değiş tokuş edilmesini sağlar. Eğitim sürecinde kullanılan YOLOv8 modeli ile yapılan deneylerde, hem CPU hem de GPU üzerinde parametre birleştirme (aggregation) işlemleri test edilmiş; GPU tabanlı senkronizasyonun %73 oranında ideal süreye yaklaşım sağladığı gözlemlenmiştir.

Ek olarak geliştirilen CPU ve GPU benchmark yazılımları sayesinde her düğümün donanımsal performansı ölçülmüş ve bu bilgiler doğrultusunda asimetrik sistemlerin birlikte çalışabilmesi mümkün kılınmıştır. Bu özellik, farklı yeteneklere sahip düğümlerin aynı eğitim sürecine dahil edilmesini sağlayarak kaynak verimliliğini üst seviyeye taşımıştır.

Bu proje, TUSAŞ'ın sahip olduğu dağınık GPU kaynaklarının bir bütün olarak kullanılmasını sağlayarak yapay zeka geliştirme süreçlerini hızlandırma ve maliyetleri düşürme potansiyeline sahiptir.

2. GENEL KISIMLAR

2.1 Yolo:

Proje, doğrudan Ultralytics tarafından geliştirilen YOLOv8 (You Only Look Once sürüm 8) nesne tanıma modeli üzerine kuruludur. YOLOv8, görüntü işleme alanında gerçek zamanlı nesne tanıma problemleri için optimize edilmiş bir yapay zeka modelidir. Model, çok katmanlı yapay sinir ağı mimarisi ile resim üzerinde nesne sınıfı ve konumunu aynı anda tespit edebilmektedir.

YOLOv8'in eğitim süreci GPU üzerinde çalışan vektörel hesaplamalara dayanmakta olup, özellikle çok boyutlu tensor veri yapıları kullanılarak gerçekleştirilir. Bu yapılar üzerinde gerçekleştirilen çarpım ve aktivasyon işlemleri GPU'lar tarafından paralel şekilde yürütülmektedir.

Projede bu modelin dağıtık eğitim ortamına uyarlanabilirliği incelenmiş, eğitim veri kümesinin batch'ler halinde farklı düğümlere dağıtılması ve ardından elde edilen sonuçların merkezî yapıda birleştirilmesi ile yüksek derecede paralellik elde edilebileceği görülmüştür. Bu açıdan YOLOv8, dağıtık mimarilere uygun hesaplama davranışı sergileyen bir model olarak değerlendirilmiştir.

2.2. PyTorch:

YOLOv8 modeli, PyTorch çerçevesi üzerinde inşa edilmiştir. PyTorch, GPU ve CPU destekli çalışabilen, tensor tabanlı matematiksel işlem altyapısı sağlayan açık kaynaklı bir derin öğrenme kütüphanesidir. Autograd, dinamik hesaplama grafiği ve yüksek esneklik gibi özellikleri ile sinir ağlarının eğitimi ve test süreçlerinde yaygın olarak kullanılmaktadır.

PyTorch, aynı zamanda dağıtık eğitim için çeşitli modüller (DDP) içermektedir. Ancak bu modüller, sistemler arası donanımsal simetri gerektirdiğinden, proje kapsamında hedeflenen asimetrik mimari ile uyumlu değildir. Simetrik yapı beklentisi nedeniyle, bu modüllerin kullanıldığı sistemlerde en yavaş düğüm tüm eğitim sürecini yavaşlatmakta ve genel verimlilik düşmektedir. Bu nedenle projede dağıtık eğitim işlemleri için, PyTorch'un iç modülleri yerine, socket tabanlı bir senkronizasyon ve veri alışverişi protokolü tasarlanmıştır.

2.3. Linux Socket :

Socket programlama, bilgisayarlar arası iletişim kurmak amacıyla işletim sisteminin sunduğu bir ağ arayüzüdür. Linux sistemlerde socket yapısı, kullanıcı düzeyinde çalışan uygulamaların ağ üzerinden veri gönderip almasına olanak tanır. Bu yapı, istemci-sunucu modeli temel alınarak geliştirilmiştir ve veri alışverişi için düşük seviyeli sistem çağrılarını kullanır.

Linux ortamında bir socket, temel olarak IP adresi ve port numarası ile tanımlanır. Uygulama tarafında, önce bir socket nesnesi oluşturulur (*socket()*), ardından bu nesneye bir adres atanır (*bind()*), bağlantılar dinlenir (*listen()*), bağlantı kabul edilir (*accept()*) ve veri gönderme/alma işlemleri (*send()*, *recv()*) gerçekleştirilir. İstemci tarafı ise *connect()* fonksiyonu ile sunucuya bağlanır.

Bu projede socket tabanlı iletişim TCP protokolü üzerinden gerçekleştirilmiştir. TCP, bağlantı odaklı (connection-oriented) bir protokol olduğu için veri bütünlüğünü ve sıralamasını garanti eder. Her istemci, eğitim süreci boyunca parametre ve kayıp fonksiyonu gibi verileri socket üzerinden sunucuya iletmekte, sunucu ise tüm istemcilerden gelen verileri toplayarak geri bildirim sağlamaktadır.

Linux socket yapısının tercih edilme nedeni, düşük seviyeli kontrol imkânı sunması ve dış kütüphane bağımlılığı olmadan doğrudan C/C++ diliyle uyumlu çalışabilmesidir. Böylece haberleşme süreci hem esnek hem de sistem kaynaklarını verimli kullanan bir yapıda tasarlanabilmiştir.

3. MATERYAL VE YÖNTEM

3.1. Sistem Değerlendirme:

3.1.1. Cluster operations:

Dağıtık eğitim yapısının sağlıklı bir şekilde çalışabilmesi için, sisteme katılacak her bir düğümün donanım özelliklerinin önceden bilinmesi gereklidir. Bu sebeple, proje kapsamında geliştirilen yapı içerisinde sistem başlangıcında yalnızca yetkin ve tanımlı düğümlerin ağa katılmasına izin verilmektedir. Bu kontrol mekanizması, Cluster Operations modülü tarafından yürütülmektedir.

Her bir istemci (düğüm), sistem başlatıldığında öncelikle kendi donanım bilgilerini (işlemci modeli, bellek miktarı, GPU tipi vb.) içeren bir sistem tanımlama paketi üretmektedir. Bu paket içerisinde, düğümün küme içerisinde hesaplamaya uygun olup olmadığını belirlemeye yönelik bilgiler yer almaktadır. Master sunucu, yalnızca bu sistem bilgisi paketini başarılı bir şekilde ileten düğümleri geçerli olarak kabul etmekte; aksi takdirde, bağlantı gerçekleşmiş olsa dahi düğüm ağa dahil edilmemektedir.

Bu yapı sayesinde, eğitim sürecine dahil olacak düğümlerin rastgele değil, belirli donanımsal koşulları sağlayan sistemlerden oluşması garanti altına alınmıştır. Sistem bilgisi paylaşılmayan veya eksik bilgi içeren düğümler, ağ güvenliği ve bütünlüğü açısından dışlanmakta; böylece eğitim sürecinde hesaplama tutarsızlıklarının ve iletişim hatalarının önüne geçilmektedir.

Bu süreçte *clusterOperations.cpp* ve *systemParser.cpp* dosyaları birlikte çalışarak, istemciden gelen sistem bilgisini ayrıştırmakta ve uygun formatta değerlendirerek kabul ya da ret kararı üretmektedir. Bu doğrulama yapılmadan hiçbir düğüm eğitim sürecine katılamamaktadır.

```
void clusterOperations::joinNodes() {
    std::thread([this] {
        std::unique_lock<std::mutex> var (std::get<0>(networkI->awaiting_connections_mtx));
        int *socket;
        while (!cleanExit) {
            if (networkI->awaiting_connections.empty())std::get<1>(networkI->awaiting_connections_mtx).wait(var);

            socket = new int;
            *socket = networkI->awaiting_connections.front();
            std::thread wait_packet([this, socket] {
                systemParser::systemNfo* snfo=nullptr;
                networkInterface::packet * pkt = nullptr;
                uint8_t packet_counter =0;
                uint8_t expectedPackets = UINT8_MAX;
                while (packet_counter < expectedPackets) {
                    pkt = networkI->searchDeserializedPacket(*socket,0x10|TYPE_SYSTEM_DATA|0x01,static_cast<uint16_t>(TIMEOUT_NODE_STATS));
                    if (pkt == reinterpret_cast<void *>(0xF0)) {
                        std::cout << "Timeout on stats of socket " << networkI->awaiting_connections.front() << ". Dropping connection" << std::endl;
                        return;
                    }
                    auto ret = systemParser::rebuildSystemNfo(snfo, pkt->opt_data,std::get<1>(pkt->data));


                    if (expectedPackets > std::get<0>(ret)) expectedPackets = std::get<0>(ret)+1; ///GPU sayisi verisinin GPU verisinden once gelecegi
                    snfo = std::get<1>(ret);
                    packet_counter++;
                }
                node* _n = new node;
                _n->nfo = snfo;
                nodes.insert({*socket,_n});
                delete socket;
            });
            wait_packet.detach();
            networkI->awaiting_connections.pop();
        }
    }).detach();
}
```

Kod bloğu-1

3.1.2. System Parser:

Dağıtık eğitim sürecinde sistemin kararlı ve verimli çalışabilmesi için, her bir düğümün donanımsal yeterliliğinin baştan belirlenmesi gerekmektedir. Özellikle eğitim görevlerinin eşit olmayan donanımlara dağıtıldığı durumlarda, işlem gücü düşük olan düğümler sistem genelinde performans kaybına yol açabilmektedir. Bu nedenle, ağa katılacak her düğümün işlemci, bellek ve grafik işlem birimi gibi temel donanım bileşenlerine ilişkin veriler, istemci tarafından sistemin başlangıç aşamasında yönetici düğüme gönderilmek üzere otomatik olarak toplanmaktadır.

Bu işlem, proje kapsamında geliştirilen *systemParser* modülü tarafından yürütülmektedir. Modül, her istemcinin çalıştığı Linux sistem üzerinde yerel terminal komutlarını çağırarak işlem yapar. Harici bir kütüphaneye ihtiyaç duymadan, işletim sisteminin sunduğu temel arayüzler üzerinden veri elde edilir. Bu yöntem, özellikle sade yapılı istemcilerde dağıtıma uygun, düşük bağımlılık içeren bir çözüm sunmaktadır.

Sistem bilgisi toplama sürecinde kullanılan yöntemler şu şekildedir:

- İşlemci bilgisi: *nproc* komutu aracılığıyla toplam çekirdek sayısı tespit edilir.
- Bellek bilgisi: RAM ve takas belleği (swap) miktarları */proc/meminfo* dosyasındaki ilgili satırlardan okunur.
- GPU bilgisi: NVIDIA ekran kartı bulunan sistemlerde, *nvidia-smi* komutu kullanılarak GPU modeli ve mevcut video belleği kapasitesi elde edilir.

Elde edilen bilgiler, sistemin hesaplama gücünü kabaca temsil eden parametrelerdir. Bu bilgiler arasında işlemcinin çekirdek sayısı, toplam ve boş RAM miktarı, ekran kartının adı ve bellek miktarı gibi öğeler yer alır. Bilgiler, sabit bir formatta derlenerek *clusterOperations* modülüne aktarılır ve burada geçerlilik kontrolü yapılır.

Düğümler bu sistem bilgisini sunucuya göndermediği sürece, bağlantı kurmuş olsalar bile ağa dahil edilmezler. Bu yaklaşım, sadece tanımlı ve uygun yapıdaki sistemlerin dağıtık eğitim ortamında yer almasını garanti eder. Böylece eğitim süreci başlatılmadan önce sistem bütünlüğü sağlanır ve hesaplama görevlerinin bilinçsiz şekilde dağılması engellenmiş olur.

systemParser modülünün terminal komutları üzerinden çalışması, platformdan bağımsızlık ve hafiflik gibi pratik avantajlar sağlar. Dış kütüphanelere olan ihtiyacı ortadan kaldırdığı için hem taşınabilirlik artar. Bu yapı, sistem genelinde hem güvenliği hem de sadeliği korurken, donanım tabanlı kararların alınmasına da temel oluşturur.

```
systemParser::systemParser() {
    collected_info = new systemNfo();
    collected_info->threadCount = getThreadCount();
    uint64_t temp_var = getMemoryNSwapSize();
    collected_info->memorySize = temp_var >> 32;
    collected_info->priority = userDefinedPriority;
    collected_info->swapSize = temp_var & 0xFFFF;
    collected_info->cpuScore = cpuBench(collected_info->threadCount);
    collected_info->availableGPUs = getGPUinfo();
    std::queue<std::thread> threads;
    if (collected_info->availableGPUs.empty())throw(std::runtime_error("No GPU available. CPU clusters not supported"));
    for (auto available_gp_u: collected_info->availableGPUs) {
        threads.emplace([*this, available_gp_u]{
                gpuBench(available_gp_u);
            std::cout << "Measured score of GPU " << +available_gp_u->id << " 16: " << available_gp_u->FP16_Score << " 32: " <<
            available_gp_u->FP32_Score << " 64: " << available_gp_u->FP64_Score <<std::endl;
            }
            );
    }

    while (!threads.empty()) {
        threads.front().join();
        threads.pop();
    }

    std::cout << "Collection of system information done." << std::endl;

    localNfo = collected_info;
}
```

Kod bloğu-2

```
uint16_t systemParser::getThreadCount(){
        uint16_t tCount = std::stoi(exec("nproc"));
    return tCount;
    }

uint64_t systemParser::getMemoryNSwapSize(){
    uint64_t memoryNswap = std::stoi(exec(R"(cat /proc/meminfo | grep -i "MemAvailable" | grep -o '[0-9]*')"));
    memoryNswap /=1024;
    memoryNswap = memoryNswap << 32;
    memoryNswap |= (std::stoi(exec(R"(cat /proc/meminfo | grep -i "SwapFree" | grep -o '[0-9]*')"))/1024);
    return memoryNswap;
}

std::vector<systemParser::gpuNfo*> systemParser::getGPUinfo() {
    std::vector<systemParser::gpuNfo*> result;
    if (exec(R"(nvidia-smi -L | grep "GPU")").length() < 2)
        return result;
    std::string str = exec(R"(nvidia-smi -L)");
    uint32_t next_line = 0, uuid_index;
    gpuNfo *gpu_nfo;
    uint8_t id_counter = 0;
    do {
        gpu_nfo = new gpuNfo();
        gpu_nfo->id = id_counter;
        next_line = str.find(':',next_line)+2;
        uuid_index = str.find('(', next_line);
        std::memcpy(gpu_nfo->name, str.c_str()+next_line , sizeof(char)*(uuid_index-7));
        std::memcpy(gpu_nfo->uuid, str.c_str()+(uuid_index+7), sizeof(char)*40);
        gpu_nfo->vram_size = std::stoi(exec(R"(nvidia-smi -i 0 -d "MEMORY" -q | grep -i "Free" | grep -o '[0-9]*' -m1)")); //memory bilgisi

        result.push_back(gpu_nfo);
        next_line = str.find('\n',uuid_index+48)!=std::variant_npos;
        id_counter++;
    } while ((next_line + MINIMAL_GPU_IDENTIFIER_SIZE) > str.length());

    return result;
}
```

Kod bloğu-3

3.1.3. CPU Ölçümü

Dağıtık sistemdeki düğümlerin hesaplama gücünün belirlenebilmesi için CPU performanslarını karşılaştırmalı olarak değerlendiren özel bir test modülü geliştirilmiştir. Bu amaçla kullanılan *cpuBench* fonksiyonu, her düğümdeki işlemcinin işlem kapasitesini zaman bazlı bir test üzerinden ölçer.

Test yapısı, paralel çalışan iş parçacıkları (thread) üzerinden yürütülür. Her bir iş parçacığı, kendisine ayrılan sürede artan sayılara bazı asal sayı kriterlerine göre bölünebilirlik testi uygular. Yani temel bir asal sayı kontrol algoritması üzerinden hesaplama yapılır. Bu hesaplamada kullanılan maksimum sayı sınırı (limit), 64-bit sistemlerin kapasitesiyle sınırlıdır. Bunu, pratik bağlamda bu limite ulaşmak mümkün olmadığı için zamana bağlı durdurma fonksiyonu kullanılır. Süre dolduğunda tüm iş parçacıkları sonlandırılır ve hesaplanan iterasyon sayıları üzerinden ortalama bir CPU skoru üretilir.

Benchmark ölçüm yöntemi olarak faktöriyel hesaplaması da denenmiş olup, faktöriyel hesabının doğası gereği sayılar çok hızlı bir şekilde büyüyebildiğinden memory management tarafında zorluklarla karşılaşılmıştır.

Test sırasında daha hızlı çalışan sistemler daha yüksek iterasyon tamamladığı için daha yüksek puan elde eder. Ayrıca işlem süresi 15 saniyenin altında tamamlanan durumlarda, elde edilen skor zamanla orantılı olarak ölçeklendirilir. Bu sayede ölçüm süresindeki dalgalanmalar minimize edilir ve daha adil bir değerlendirme sağlanır.

Sonuç olarak elde edilen CPU skoru, her düğümün sistem bilgileriyle birlikte sunucuya iletilir ve sistemin hesaplama görevlerini dağıtma sürecinde temel parametrelerden biri olarak kullanılır. Böylece, farklı donanım seviyelerine sahip düğümlerin kapasitesi karşılaştırılabilir hale getirilmiş olur.

```
void primeCheck(uint64_t *p) {
    bool isPrime = true;
    uint64_t i = 0;
    double start = cpu_read_timer();
    for (i = 2; i <= SIZE_LIMIT; i++) {
        if (kill) break;
        if (i%2 == 0)isPrime=false;
        else if (i%3 == 0)isPrime=false;
        else if (i%5 == 0)isPrime=false;
        else if (i%7 == 0)isPrime=false;
    }
    double time_taken = cpu_read_timer()-start;

    if (static_cast<unsigned int>(time_taken) < TIME_LIMIT) i *= TIME_LIMIT /static_cast<unsigned int>(time_taken);
#ifdef DEBUG
    std::cout << +time_taken << "Thread id: " << gettid() << std::endl;
#endif

    *p = i;
}
```

Kod bloğu-4

```
#define SIZE_LIMIT UINT64_MAX
#define TIME_LIMIT 15
```

Kod bloğu-5

```
uint32_t cpuBench(uint16_t &threadCount){

    std::queue<std::thread> threads;
    auto results = new uint64_t[threadCount];

    std::cout << "Starting cpu benchmark with " << +threadCount << " workers" << std::endl;
    for (uint16_t i = 0; i < threadCount; i++) {
        threads.push(std::thread(primeCheck, &results[i]));
    }

    std::this_thread::sleep_for(std::chrono::seconds(TIME_LIMIT));
    kill = true;

    while (threads.size() > 0) {
        threads.front().join();
        threads.pop();
    }

    uint64_t result = 0;
    for (uint16_t i = 0; i < threadCount; i++) {
        result += results[i];
    }

    result /= threadCount;
    result = result >> 16;

std::cout << "Measured cpu score: " << result << std::endl;

    return static_cast<uint32_t>(result);

}
```

Kod bloğu-6

3.1.4. GPU Ölçümü

GPU'lar, paralel işlem kapasitesi sayesinde derin öğrenme eğitimlerinde merkezi rol oynar. Ancak her GPU'nun işlem gücü, kullanılan precision' a bağlı olarak farklılık göstermektedir. Bu nedenle, sistemde yer alan her GPU'nun yarı hassasiyetli (FP16), tek hassasiyetli (FP32) ve çift hassasiyetli (FP64) işlemler için ayrı ayrı performans değerleri ölçülmüştür.

Bu değerlendirme *gpuBench.cu* modülü tarafından yapılmaktadır. Modül, her bir tür için iki adet rastgele matris oluşturur ve bu matrisler çarpılarak elde edilen işlemin süresi ölçülür. Süre bilgisi, ilgili hassasiyet türündeki işlem kapasitesini temsil eden puana dönüştürülür ve diğer düğümlerle karşılaştırmada kullanılmak üzere saklanır.

Her sayı türü için ayrı ölçüm yapılmasının temel nedeni, farklı GPU mimarilerinin bu türleri farklı hızlarda işlemesidir.

Örneğin:

- FP64 işlemler, kullanıcı sınıfı ekran kartlarında, örneğin RTX 4090 Ti için bile, genellikle FP32'ye kıyasla 8-9 kat daha yavaştır.

- FP16 (half precision) işlemler ise çoğu GPU'da FP32'ye göre yaklaşık 2 kat daha hızlı çalışabilir; ancak bu oran her GPU'da sabit değildir.
- Bazı sunucu sınıfı GPU'lar FP64 hesaplamaları için özel donanım içerdiğinden, burada FP64 performansı FP32 ile aynı seviyeye kadar çıkabilir.

Bu nedenle, her GPU'nun hangi sayı türlerinde ne kadar verimli olduğu önceden ölçülmeli ve düğümlerin eğitim görevlerinde hangi formatta kullanılacağı bu verilere göre optimize edilmelidir. Özellikle YOLOv8 gibi FP16 üzerinde çalışan modeller için, yarı hassasiyetli işlemlerde yüksek performans sağlayan GPU'lar tercih edilmektedir. Öte yandan, bazı matematiksel işlemlerde yüksek doğruluk gerektiği için FP64 de kritik bir değerlendirme parametresi olarak yer almaktadır.

gpuBench modülü bu bağlamda, dağıtık yapının heterojenliğini tanıma ve görev dağılımı yaparken donanıma duyarlı bir karar mekanizması oluşturma açısından önemli rol oynamaktadır. Elde edilen sonuçlar, sistemdeki her GPU için ayrı ayrı FP16, FP32 ve FP64 skorlarını içerecek şekilde kayıt altına alınmaktadır.

```
template <class precision>
bool initOp(int &size, cublasStatus_t &stat, cublasHandle_t&handle, precision &one, precision &zero, precision * &A, precision * &B, double &tInit,
  precision *&dA, precision *&dB, int &value1) {
  cudaError_t cudaStat;
  int it;

  cublasOperation_t N = CUBLAS_OP_N;
  cublasOperation_t T = CUBLAS_OP_T;
  one = 1.;
  zero = 0.;

  size = 8192*2;

  A = (precision*) malloc( sizeof(precision)*size*size );
  B = (precision*) malloc( sizeof(precision)*size*size );

  cudaDeviceSynchronize();
  tInit = read_timer();

  // allocate memory on device (GPU)
  cudaStat = cudaMalloc((void**)&dA, sizeof(precision)*size*size);
  if(cudaStat != cudaSuccess) {
    printf ("device memory allocation failed");
    value1 = EXIT_FAILURE;
    return true;
  }
  cudaStat = cudaMalloc((void**)&dB, sizeof(precision)*size*size);
  if(cudaStat != cudaSuccess) {
    printf ("device memory allocation failed");
    value1 = EXIT_FAILURE;
    return true;
  }
  return false;
}
```

Kod bloğu-7

```cpp
int gpuBench(systemParser::gpuNfo *nfo) {
  cudaSetDevice(nfo->id);
  printf("Starting benchmark for GPU ");
  std::cout << +nfo->id << std::endl;
  auto float_res = createOperationsS<float>();
  printf("Stage 1/3\n");
  nfo->FP32_Score = static_cast<uint16_t>(UINT16_MAX / (std::get<1>(float_res)*SCORE_MULTIPLIER));
  printf("Stage 2/3\n");
  nfo->FP64_Score = static_cast<uint16_t>(UINT16_MAX / (createOperationsD<double>()*SCORE_MULTIPLIER));
  printf("Stage 3/3\n");
  nfo->FP16_Score = static_cast<uint16_t>(UINT16_MAX / (createOperationsH<__half>()*SCORE_MULTIPLIER));
  nfo->dataLoad_Score = std::get<0>(std::get<0>(float_res));
  nfo->dataOffload_Score = std::get<1>(std::get<0>(float_res)); //fixme: dataOffload skoru 0 oluyor
  return EXIT_SUCCESS;
}
```

Kod bloğu-8

3.1.4. Scheduling:

Dağıtık sistem yapılarında temel problemlerden biri, her düğümün işlem gücünün farklı olması durumunda iş yükünü adil ve verimli bir şekilde paylaştırmaktır. Projenin bu kısmındaki yapı içerisinde, düğümlere atanacak görev miktarı, hem benchmark skorlarına hem de sistem performansına bağlı olarak dinamik şekilde belirlenmektedir.

Bu süreçte ilk olarak, sistemde bulunan her düğümün yapısal uygunluğu denetlenir. Yani "belirli bir fonksiyon veya işlem tipi o düğüm tarafından destekleniyor mu?" gibi. Eğer desteklenmiyorsa, ilgili düğüm hesaplamaya dahil edilmez; aksi takdirde tüm küme içinde işlem bozulur. Bu durumda sistemin istikrarını sağlamak adına, ya söz konusu düğüme görev verilmez, ya da yapı dışına çıkarılır.

İlgili düğümler filtrelendikten sonra, her düğüm için bir maliyet fonksiyonu hesaplanır. Bu fonksiyon, temel olarak veri büyüklüğü, ağ gecikmesi, sistem belleği ve işlem süresi gibi değişkenleri dikkate alır. Aşağıda kullanılan formül gösterilmiştir:

Maliyet Fonksiyonu:

$$\underbrace{\frac{\left((L_{load} + L_{receive}) * \lceil \frac{size_{data}}{size_{memory}} \rceil\right) + T}{\frac{w}{\lambda} * (size_{data} (\text{mod } size_{memory}))}}_{w \neq 0 \; ve \; \lambda < 0,25}$$

$$\lambda = \left(\frac{L_{load} + L_{receive}}{T} \right)$$

$$T = t_{instruction} * \left(\prod_{d=1}^{dim} |data_d| \right)$$

$T_{instruction}$, gerçeklenecek emirin aldığı birim zamanı

dim, verinin boyut sayısını

T, hesaplamanın alacağı zamanı

L_{load}, yükleme gecikmesini

$L_{receive}$, alma gecikmesini

λ, ağ transferinden kaynaklanan gecikmenin işlem süresine oranını

$size_{data}$, veri boyutunu

$size_{memory}$, müsait bellek boyutunu

w, fonksiyonun ağırlılandırmasını belirler.

Denklem-1: Loss fonksiyonu

Elde edilen her bir maliyet değeri, benchmark skorlarıyla birlikte değerlendirilir. Maliyet fonksiyonu yüksek çıkan düğümler, sistemde daha yavaş ve verimsiz kabul edilir. Dolayısıyla daha az iş yükü alır. Bu bağlamda her bir düğüme atanacak batch sayısı, benchmark skorları ve gecikme maliyeti oranlanarak belirlenir.

Örneğin:

- Düşük maliyetli bir RTX 4090 Ti düğümüne 8 batch
- Daha zayıf bir düğüme 4 batch
- Daha düşük kapasiteli bir düğüme ise yalnızca 2 batch verilebilir

Ancak bu batch'ler tek tek gönderilmek yerine, mümkün olduğunca VRAM sınırları içinde gruplanarak gönderilir. Böylece daha az aktarım yapılır ve genel sistem gecikmesi düşer. Bu işlem, en büyük ortak bölen (EBOB) üzerinden yapılan oranlama ile batch gruplarının birleştirilmesini mümkün kılar.

Sonuç olarak planlanan bu yapı, hem sistemin donanımsal dengesizliğini göz önünde bulundurarak adil görev dağılımı yapılmasını, hem de ağ ve bellek optimizasyonları sayesinde işlem veriminin artırılmasını hedeflemektedir. Bu yönüyle sadece benchmark skorlarına değil, gerçek zamanlı iletişim ve hesaplama koşullarına da tepki veren esnek bir yük dengeleme stratejisi sunmaktadır.

3.2. Düğümler arası bağlantı mekanizması:

Dağıtık sistem mimarisinde, düğümler arası bağlantı TCP tabanlı soket yapısı ile kurulmaktadır. Her istemci, merkezi bir sunucuya bağlanır ve eğitim süreci boyunca veri alışverişi bu bağlantılar üzerinden gerçekleştirilir.

3.2.1. İstemci (client) yapısı

Projede istemci tarafındaki bağlantı mekanizması, her bir düğümün belirli bir IP adresi ve port üzerinden merkezi sunucuya ulaşması prensibine dayanır. Bağlantı sağlandıktan sonra sistem, çift yönlü (duplex) iletişim kurulacak şekilde yapılandırılmıştır. Bu yapı sayesinde istemci, yalnızca sunucudan veri almakla kalmaz; aynı zamanda kendi tarafındaki hesaplama sonuçlarını da aktif olarak sunucuya iletebilir.

İletişim süreci, istemci tarafında arka planda sürekli çalışan bir döngüyle yürütülür. Bu döngü, bir yandan sunucudan gelen veri paketlerini dinlerken, diğer yandan sistemdeki parametre güncellemelerini veya kontrol mesajlarını geri göndermekle sorumludur. Bu çift yönlü yapı, dağıtık eğitim sürecinin senkronizasyonunu sağlamak açısından kritik öneme sahiptir.

Bağlantı kurulduktan sonra istemci, eğitim çıktısı olan parametreleri ve kayıp fonksiyonuna ait değerleri sunucuya aktarır. Ardından, diğer düğümlerden gelen verilerin birleşimiyle oluşturulan güncel model parametrelerini alarak kendi tarafındaki eğitim sürecine devam eder. Bu döngü, her eğitim adımı sonunda tekrarlanır.

```python
def create_packet(opt_data: bytes, data: bytes, opt_usage: int) -> bytes:  1 usage
    has_opt = len(opt_data) > 0
    opt_size = len(opt_data)
    data_size = len(data)

    if data_size == 0 and opt_size == 0:
        return struct.pack( fmt: "B",  *v: TYPE_EMPTY)

    if not has_opt and data_size > 0:
        return struct.pack( fmt: "B",  *v: TYPE_PLAIN_DATA) + struct.pack( fmt: "I",  *v: data_size) + data

    type_byte = make_type_byte(has_opt, opt_size, opt_usage)
    return struct.pack( fmt: "B",  *v: type_byte) + opt_data + struct.pack( fmt: "I",  *v: data_size) + data
```

Kod bloğu-9

```
def recv_loop(self):  1 usage
    while self.running:
        try:
            first_byte = self._recv_exact(1)[0]

            if is_nac_request(first_byte):
                second_byte = self._recv_exact(1)[0]
                info = parse_nac_request(bytes([first_byte, second_byte]))
                print(f"[RECV] NAC_REQUEST: {info}")
                continue
            elif is_nac_status(first_byte):
                info = parse_nac_status(first_byte)
                print(f"[RECV] NAC_STATUS: {info}")
                continue
            elif is_nac_pico(first_byte):
                info = parse_nac_pico(first_byte)
                opt_data = self._recv_exact(info['size'])
                data_size = struct.unpack( format: "I", self._recv_exact(4))[0]
                data = self._recv_exact(data_size)
                print(f"[RECV] NAC_PICO: {info}, opt={opt_data}, data={len(data)} bytes")
                continue
            elif first_byte == TYPE_NAC_PRINT:
                msg = b""
                while True:
                    ch = self._recv_exact(1)
                    if ch == b"\x00" or not ch:
                        break
                    msg += ch
                print(f"[RECV] NAC_PRINT: {msg.decode(errors='ignore')}")
                continue
            elif first_byte == TYPE_EMPTY:
                print("[RECV] EMPTY packet")
                continue
            elif first_byte == TYPE_PLAIN_DATA:
                size = struct.unpack( format: "I", self._recv_exact(4))[0]
                data = self._recv_exact(size)
                print(f"[RECV] PLAIN DATA: {len(data)} bytes")
                continue

            has_opt, opt_size, usage = parse_type_byte(first_byte)
            opt_data = self._recv_exact(opt_size) if has_opt else b''
            data_size = struct.unpack( format: "I", self._recv_exact(4))[0]
            data = self._recv_exact(data_size)
            print(f"[RECV] PICO: opt={opt_data}, data={len(data)} bytes")

        except Exception as e:
            print("Receive error:", e)
            self.running = False
```

Kod bloğu-10

```
    def send_loop(self):  1 usage
        while self.running:
            try:
                packet = self.send_queue.get(timeout=1)
                self.sock.sendall(packet)
                self.send_queue.task_done()
                print(f"[SEND] Sent {len(packet)} bytes")
            except queue.Empty:
                continue
            except Exception as e:
                print("Send error:", e)
                self.running = False

    def send_data(self, opt_data: bytes, data: bytes, usage_type: int = 0):  2 usages
        packet = create_packet(opt_data, data, usage_type)
        self.send_queue.put(packet)

    def send_nac_print(self, msg: str):  1 usage
        msg_bytes = msg.encode('utf-8') + b'\x00'
        packet = struct.pack( fmt: "B",  *v: TYPE_NAC_PRINT) + msg_bytes
        self.send_queue.put(packet)

    def stop(self):  1 usage
        print("[CLIENT] Waiting for queue to flush...")
        self.send_queue.join()
        print("[CLIENT] Flushed. Closing socket.")
        self.running = False
        self.sock.close()

    def _recv_exact(self, n):  11 usages
        buf = b''
        while len(buf) < n:
            chunk = self.sock.recv(n - len(buf))
            if not chunk:
                raise ConnectionError("Socket closed")
            buf += chunk
        return buf
```

Kod bloğu-11

3.2.2. Sunucu Yapısı

Sunucu tarafı, yine *networkInterface* sınıfı ile tanımlanır ve belirtilen port üzerinden bağlantı dinleme işlemini başlatır. *listen_n_bind* fonksiyonu içerisinde gelen bağlantılar şu adımlarla sisteme dahil edilir:

- Yeni bağlantıya ait soket dosya tanıtıcısı (descriptor) kaydedilir.
- Gerekli senkronizasyon yapıları (mutex, condition variable) bu sokete atanır.
- Veri alımı ve çözümlemesi için iki ayrı iş parçacığı başlatılır.

- Bağlantı doğrulaması yapılır ve istemci sistem bilgisi göndermediği sürece aktif sayılmaz

3.2.3. Duplex iletişim gerekçesi

Sistem, duplex bir iletişim modeliyle tasarlanmıştır. Bu yapıda, hem istemciler hem de sunucu birbirine aktif olarak veri gönderip alabilmektedir.

Bu tercih edilmiştir çünkü:

- Eğitim sürecinde her istemcinin modeli eğitip elde ettiği parametreleri sunucuya göndermesi gerekir.
- Aynı zamanda sunucunun güncellenmiş model parametrelerini istemcilere tekrar dağıtması gerekir.

Bu çift yönlü veri akışı, sadece belirli zamanlarda değil, sürekli ve senkronize bir biçimde gerçekleşir. Bu nedenle sistem, pasif bekleyen bir yapı yerine, sürekli dinleyen ve gönderen paralel iş parçacıklarına sahip bir duplex bağlantı modeliyle tasarlanmıştır. Bu yapı sayesinde sistem:

- Gerçek zamanlı güncelleme yapılabilmekte.
- Her düğüm senkronize şekilde eğitim sürecine katılabilmekte.
- Parametre paylaşımı aksamadan sürdürülebilmektedir.

3.3. Hesaplamanın Gerçekleştirilmesi ve Kombinasyonu

3.3.1. Hesaplamanın Gerçekleştirilmesi

TUSAŞ tarafından kullanılan YOLOv8 modeli, bizim tarafımızdan geliştirilmemiş bir açık kaynak yapıdır. Bu nedenle modelin eğitim sürecine müdahale edebilmek için klasik bir entegrasyon yaklaşımı yerine, projemizin en uzun süren kısmı olan, tersine mühendislik yöntemine başvurulmuştur. Amaç, modelin hangi aşamalarında hangi parametrelerin değiştiğini tespit ederek, eğitim süreci içerisinde dağıtık kontrol noktaları oluşturmaktır.

İlk aşamada, YOLOv8 modelini çalıştıran eğitim süreci, GDB (debugger) kullanılarak her bir instruction izlenmiştir. Bu süreçte, eğitim başladığında modelin sırasıyla hangi fonksiyonlara girdiği tespit edilmiştir. Takip sonucunda modelin ağırlıklarının (weights), bias terimlerinin ve metriklerinin (precision, recall, mAP vb.) güncellendiği fonksiyonun do_train olduğu belirlenmiştir. Bu gözlem, eğitim sürecinin temel mantığının bu fonksiyon içinde gerçekleştiğini açıkça ortaya koymuştur.

Bu tespitin ardından, *do_train* fonksiyonu içerisindeki tüm değişkenler kopyaları alınıp, karşılaştırılıp, takip edilerek her bir epoch ve batch sırasında nasıl değiştikleri analiz edilmiştir. Sabit kalan değişkenler dışlanmış ve yalnızca değişim gösteren parametreler detaylı biçimde incelenmiştir. Bu yaklaşım, eğitim sürecinde kritik olan değişkenlerin belirlenmesini mümkün kılmıştır.

```
iter_accumulate = deepcopy(self.accumulate)  #ayni
iter_amp = deepcopy(self.amp)  #ayni
iter_args = deepcopy(self.args)  #ayni
iter_batch_size = deepcopy(self.batch_size)  # //Ayni
iter_best = deepcopy(self.best)  # //Ayni
iter_best_fitness = deepcopy(self.best_fitness)  # //Ayni
iter_csv = deepcopy(self.csv)  # //Ayni
iter_data = deepcopy(self.data)  # //Ayni
iter_device = deepcopy(self.device)  # //Ayni
iter_ema = deepcopy(self.ema)  #DEGISTI | ema.model:DEGISTI
iter_last = deepcopy(self.last)  # //Ayni
iter_loss_names = deepcopy(self.loss_names)  # //Ayni
iter_loss = copy(self.loss)  #true dondu ama false olan parametreler de var
iter_metrics = deepcopy(self.metrics)  # //Ayni
iter_model = deepcopy(
    self.model)  #DEGISTI | .criterion:DEGISTI, .model:DEGISTI,
iter_optimizer = deepcopy(
    self.optimizer)  #DEGISTI
iter_plot_idx = deepcopy(self.plot_idx)  # //Ayni
iter_plots = deepcopy(self.plots)  # //Ayni
iter_save_dir = deepcopy(self.save_dir)  # //Ayni
iter_scaler = deepcopy(self.scaler)  #DEGISTI | ._growth_tracker:DEGISTI, ._scale:DEGISTI
iter_scheduler = deepcopy(
    self.scheduler)  #DEGISTI ||| !!!self.scheduler.optimizer ESITTIR self.optimizer!!! |||
iter_start_epoch = deepcopy(self.start_epoch)  # //Ayni
iter_stopper = deepcopy(self.stopper)  #DEGISTI
iter_test_loader_dataset = deepcopy(self.test_loader.dataset)
iter_tloss = deepcopy(self.tloss)  # true dondu ama false parametreler de var
iter_train_loader_dataset = deepcopy(self.train_loader.dataset)  # //Gorunuste ayni
iter_train_time_start = deepcopy(self.train_time_start)  # //Ayni
```

Kod bloğu-12 (Model Değişkenleri)

Süreç boyunca yapılan gözlemler sonucunda, optimizer aşamasında BatchNorm2d katmanlarının önemli bir rol oynadığı fark edilmiştir. Bu yapı, her batch'in dağılımını normalize ederek, öğrenme sürecinin kararlılığını artırmayı amaçlayan bir işlemdir. Söz konusu katmanın davranışını daha iyi anlamak adına, PyTorch dokümantasyonunda belirtilen ilgili bilimsel yayın (paper) incelenmiş ve algoritmanın temel prensipleri öğrenilmiştir.

Elde edilen bu bilgiler doğrultusunda, iki farklı bilgisayarda paralel olarak çalışan YOLO modellerinden eğitim sırasında BatchNorm2d sonrasındaki model parametreleri ve kayıp (loss) değerleri alınmıştır. Loss değerlerinin zaten normalize edilmiş yapıda olması sayesinde, Model matriksi boyuncaki kernel-windowda mean değeri alınarak iki farklı sistemin ortak bir eğitim sürecine dahil edilmesi sağlanmıştır.

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

Denklem-2: batchNorm2D [5]

Bu işlem, modelin eğitim sürecinde senkronizasyonun temelini oluşturmuş; merkezi sunucunun her batch sonrasında gelen ağırlıkları ve loss değerlerini ortalamasıyla güncellenmiş model

tekrar istemcilere gönderilerek eğitim devam ettirilmiştir. Böylece, modelin eğitimi fiziksel olarak ayrı bilgisayarlarda yapılmasına rağmen, tek bir model gibi davranması sağlanmıştır.

3.3.2. Model Kombinasyonu

Model eğitimi her bir istemcide ayrı ayrı gerçekleştirildikten sonra, merkezi sunucu bu düğümlerden gelen eğitim çıktıları ile ortak bir model üretmek üzere parametre birleştirme (kombinasyon) işlemini gerçekleştirmektedir. Bu süreçte her istemci, kendi üzerinde eğittiği modelin güncellenmiş ağırlık ve bias'larını (*state_dict*) ve eğitim sonucunda elde ettiği loss değerini sunucuya iletir.

Merkezi sunucu, bu ağırlıkları aldıktan sonra belirli birleştirme stratejisi ile ortak bir model üretir. Bu işlem, sistem mimarisine göre hem CPU hem de GPU üzerinde denenmiş ve karşılaştırılmıştır.

3.3.2.1. İterasyon 1: CPU Tabanlı Kombinasyon

İlk denemelerde, model ağırlıklarının birleşimi CPU üzerinde gerçekleştirilmiştir. İstemcilerden gelen parametreler dönüştürülerek her bir tensör için aritmetik ortalama alınmış ve birleşik model parametreleri oluşturulmuştur. Yöntem işlevsel olmakla birlikte, parametrelerin sıralı olarak işlenmesi ve bellekte taşınması nedeniyle zaman açısından sınırlayıcı olmuştur.

```
def thread_list(self, benkimim):
    now = datetime.datetime.now()
    print(f"thread {benkimim},{now}")
    for i, key in enumerate(self.state_key_list):

        if i % 6 == benkimim:

            try:
                self.model.model.state_dict()[key] += (self.karsi_model[key])
                if "tracked" not in key:
                    self.model.model.state_dict()[key] /= 2
            except RuntimeError:
                print("Beni debugla")

    now = datetime.datetime.now()
    print(f"thread {benkimim}, bitti {now}")
```

Kod bloğu-13

3.3.2.2. İterasyon 2: GPU Tabanlı Kombinasyon

Performansın artırılması amacıyla ağırlık birleştirme işlemi alternatif bir yöntem olarak GPU üzerinde uygulanarak karşılaştırılmıştır. Bu işlemde, her bir model parametresi doğrudan GPU belleğinde *add* ve *div* işlemleri ile birleştirilmiş ve tüm tensörler paralel olarak işlenmiştir. Eğitim dışı bazı takip verileri bu işleme dahil edilmemiştir.

GPU belleği üzerinden gerçekleştirilen bu birleştirme işlemi, özellikle yüksek boyutlu modeller ve çok sayıda istemciyle çalışıldığında zaman açısından daha verimli sonuçlar vermiştir. Bu nedenle sistemin sonraki sürümlerinde parametre kombinasyonu GPU üzerinde gerçekleştirilmiş ve eğitim döngüsü bu yapıya göre optimize edilmiştir.

Model parametreleri birleştirildikten sonra güncellenmiş haliyle istemcilere tekrar gönderilmiş ve bir sonraki eğitim döngüsüne geçilmiştir. Bu süreç, modelin merkezi olarak güncellenmesini ve tüm düğümlerin ortak bir modeli senkronize biçimde eğitmesini mümkün kılmıştır.

```
def parallell_update(self,  1 usage
                     model: torch.nn.Module,
                     karsi_model_state: OrderedDict,
                     state_key_list: list[str],
                     benkimim: int,
                     step: int = 6):

    with torch.no_grad():
        sd = model.model.state_dict()


        for i in range(len(state_key_list)):
            key = state_key_list[i]
            param = sd[key]
            other = karsi_model_state[key]

            # 3) In-place toplama ve bölme
            param.add_(other)
            if "tracked" not in key:
                param.div_(2)
```

Kod bloğu-14

4. BULGULAR

Sistem performansı, aynı veri seti üzerinde farklı yapılandırmalarda eğitilen YOLOv8 modelinin epoch süreleri karşılaştırılarak değerlendirilmiştir. Eğitimler üç ayrı senaryo altında yürütülmüştür:

- Tek bilgisayar (yerel eğitim)
- İki bilgisayar (CPU ile parametre kombinasyonu)
- İki bilgisayar (GPU ile parametre kombinasyonu)

Her üç durumda da 23 epoch boyunca eğitim gerçekleştirilmiş ve her epoch süresi kaydedilmiştir. Epoch süreleri aşağıdaki gibidir:

epoch	Tek Bilgisayarda Süreler(s)	İki Bilgisayarda Süreler (CPU Kombinasyon-threaded)(s)	İki Bilgisayarda Süreler (GPU Kombinasyon) (s)
1	9.92604	10.8425	14.0834
2	19.2874	18.4367	20.7046
3	28.6643	26.1296	27.1573
4	38.1522	33.9683	34.8032
5	47.3838	41.6184	41.393
6	56.8153	49.1502	47.9276
7	65.953	56.8186	54.3281
8	75.3654	64.4597	60.7544
9	84.9608	72.0889	67.1271
10	95.3039	79.825	73.6784
11	105.658	87.4372	80.174
12	115.423	95.0358	86.7321
13	126.069	102.609	93.2505
14	136.399	110.24	99.6533
15	145.93	117.788	106.303
16	155.969	125.39	112.6
17	166.333	133.475	119.101
18	175.957	141.885	125.763
19	186.284	150.179	132.349
20	196.591	157.968	138.734
21	206.392	165.519	145.333
22	216.656	173.03	152
23	227.063	180.712	158.626

Tablo-1: Epoch süreleri

Dağıtılmış sistem için ideal süre, tek bilgisayarda yapılan eğitim süresinin yarısı olarak kabul edilmiştir. Buna göre Tablo-1 de en son epoch' taki verilere göre hesaplayacak olursak ideal süreye yaklaşımımız:

Bir bilgisayarın eğitim süresinin yarısı;

$$\frac{227.06}{2} = 113.53\text{s}$$

İki bilgisayarlı sistemde program başlatılırken geçen ek 4 saniye çıkarıldığında;

$$158.62 - 4 = 154.62\text{s}$$

İdeal süreye yaklaşım yüzdesi;

$$\frac{113.53}{154.62} \times 100 = 73.42\%$$

Olarak bulunur.

Bu eğitime ait diğer performans metrikleri "EKLER" bölümündedir.

5. TARTIŞMA VE SONUÇ

Bu çalışmada, yapay zeka modellerinin dağıtık ortamlarda eğitilmesini sağlayan, çoklu GPU'lara sahip düğümler arasında ağ üzerinden haberleşme kurabilen bir sistem geliştirilmiştir. Amaç, TUSAŞ bünyesinde mevcut olan çeşitli donanımların bir araya getirilerek daha verimli kullanılması ve yüksek işlem gücü gerektiren modellerin eğitim sürelerinin azaltılmasıdır.

Testler, YOLOv8 modeli üzerinden yürütülmüş ve elde edilen veriler, dağıtık eğitimin doğru yapılandırıldığında tek düğümde yürütülen eğitime kıyasla daha kısa sürede tamamlanabildiğini göstermiştir. Parametrelerin birleştirilmesi için hem işlemci hem de grafik işlemci tabanlı yaklaşımlar kullanılmış, özellikle grafik işlemci üzerinden yapılan birleştirmelerde daha yüksek zaman tasarrufu sağlanmıştır. Bu nedenle GPU destekli yöntem sistemin tercih edilen çalışma biçimi olmuştur ve bu yöntemle %73'e varan ideale yaklaşım hesaplanmıştır.

Farklı düğümlerin işlem gücü ve bellek kapasiteleri gibi sistem bilgilerini alan yazılım geliştirilmiş, bu bilgiler doğrultusunda yalnızca uygun koşulları sağlayan düğümlerin hesaplama sürecine dahil edilmesini mümkün kılmıştır. Ayrıca, her düğümün iş yükü kapasitesi net olarak belirlenerek kaynakların verimli dağıtılması mümkün olmuştur.

Bununla birlikte sistemin bazı sınırlamaları da gözlemlenmiştir. Eğitim süreci tüm istemcilerde eşzamanlı yürütüldüğünden, herhangi bir düğümde yaşanan gecikme veya kopma tüm eğitim sürecini etkilemektedir. Ayrıca model parametrelerinin her eğitim döngüsü sonunda tüm düğümler arasında paylaştırılması, veri trafiğini artırmakta ve ağ gecikmesine yol açabilmektedir. Bu etkiler, düğüm sayısı ve model karmaşıklığı arttıkça daha belirgin hale gelmektedir.

Sonuç olarak geliştirilen sistem, farklı donanımlara sahip makinelerin ağ üzerinden işbirliği içinde çalışarak yapay zeka model eğitimi gerçekleştirmesini mümkün kılmıştır. Mevcut kaynakları bir araya getirerek daha düşük maliyetli bir dağıtık hesaplama ortamı oluşturulmuştur. Sistem, farklı altyapılara uyarlanabilir yönüyle taşınabilir bir çözüm sunmakta; aynı zamanda ileri düzey yapay zeka uygulamalarında kullanılabilecek esnek bir temel altyapı oluşturmaktadır.

Bu proje, TUBİTAK 2209B kapsamında geliştirilmeye devam edilecektir.

Çalışma boyunca:

- Daha kolay kullanım için arayüz geliştirilebilir.
- Yönetici düğümün düşmesi durumunda kümenin dağılmaması için yedeklilik mekanizması geliştirilebilir.
- Hesaplama esnasında düğüm düşmesi hâlinde hesaplamanın iptal olmaması için geçici bir tabloda hesaplama verileri tutulabilir.

KAYNAKLAR

[1] “Machine Learning Examples, Applications & Use Cases | IBM,” www.ibm.com, Jul. 25, 2024. https://www.ibm.com/think/topics/machine-learning-use-cases

[2] N. Thompson, K. Greenewald, K. Lee, and G. Manso, “THE COMPUTATIONAL LIMITS OF DEEP LEARNING,” 2020. Available: content/uploads/2020/09/RBN.Thompson.pdf

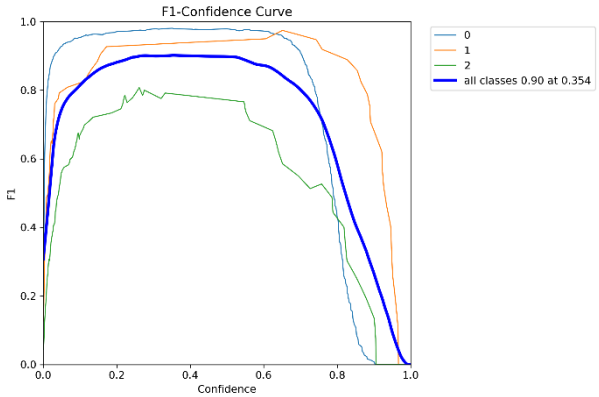
[3] K. D. Underwood, R. R. Sass and W. B. Ligon, "Cost Effectiveness of an Adaptable Computing Cluster," SC '01: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing, Denver, CO, USA, 2001, pp. 30-30, doi: 10.1145/582034.582088.

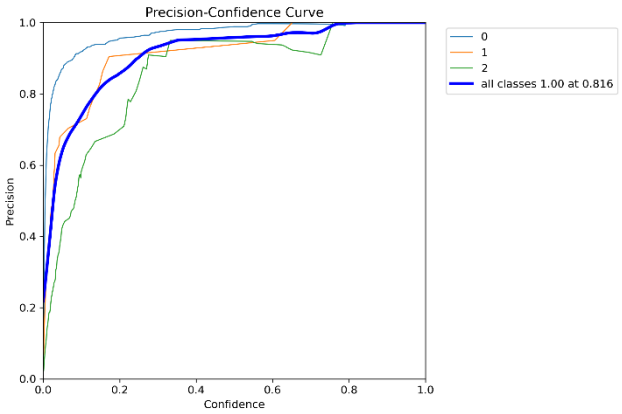
[4] Y. Ou, H. Chen and L. Lai, "A dynamic load balance on GPU cluster for fork-join search," 2011 IEEE International Conference on Cloud Computing and Intelligence Systems, Beijing, China, 2011, pp. 592-596, doi: 10.1109/CCIS.2011.6045138.

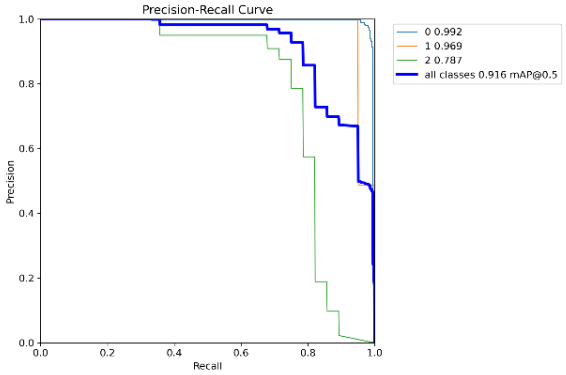
[5] Ioffe, S. (2015) Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. Available at: https://arxiv.org/pdf/1502.03167.

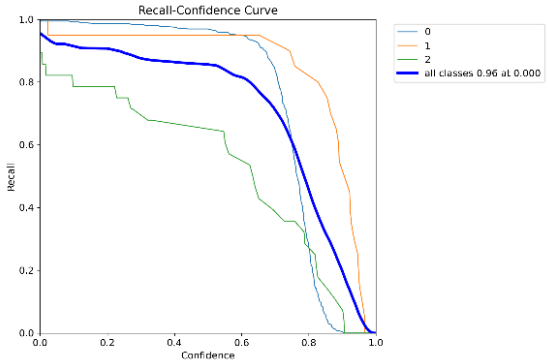
EKLER

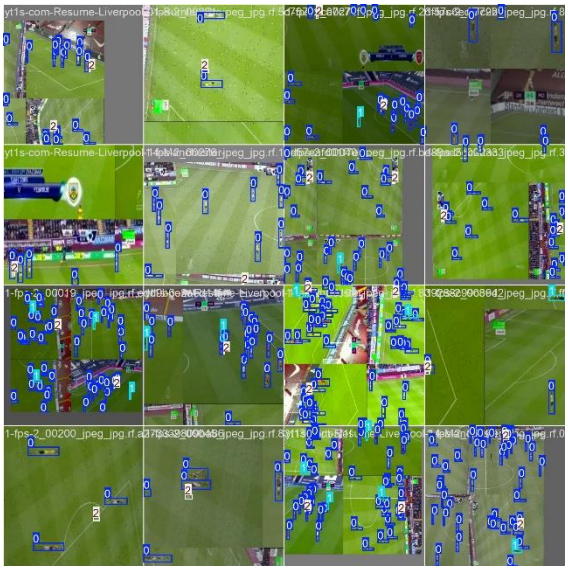
Ek-1: Eğitim metrikleri



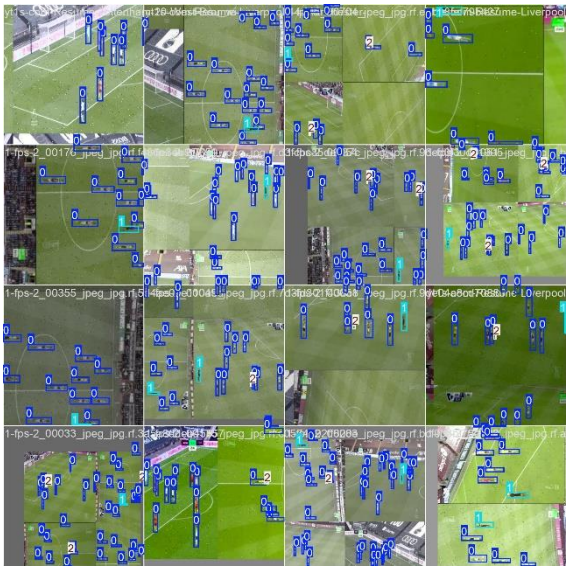








Görsel 1-Training sonucu label'lar



Görsel 2- Training sonucu label'lar

epoch	time	train/box_loss	train/cls_loss	train/dfl_loss	metrics/precision(B)	metrics/recall(B)	metrics/mAP50(B)	metrics/mAP50-95(B)	val/box_loss	val/cls_loss	val/dfl_loss	lr/pg0	lr/pg1	lr/pg2
1	10.1578	2.09364	3.76338	1.35929	0.03003	0.25223	0.11905	0.05682	1.77217	3.51879	1.09738	0.0001286	0.0001286	0.0001286
2	18.2227	1.81931	2.57799	1.12792	0.0341	0.42493	0.22249	0.08372	1.76292	3.14438	1.13795	0.0003762	0.0003762	0.0003762
3	24.8205	1.71459	1.60719	1.09247	0.03543	0.52851	0.19729	0.05793	2.02464	3.37097	1.30336	0.0006121	0.0006121	0.0006121
4	32.4667	1.68094	1.38973	1.14076	0.03219	0.39327	0.11272	0.03436	2.02386	3.30259	1.42522	0.0008334	0.0008334	0.0008334
5	39.0225	1.59343	1.25859	1.12141	0.03963	0.54712	0.2894	0.13578	1.63587	2.92335	1.16366	0.0010429	0.0010429	0.0010429
6	46.5435	1.59305	1.22516	1.13438	0.03953	0.36003	0.25577	0.08985	1.89839	2.9844	1.3287	0.0012396	0.0012396	0.0012396
7	53.1087	1.62609	1.17728	1.12463	0.01351	0.15304	0.06019	0.01601	2.91173	3.13502	1.8498	0.0012167	0.0012167	0.0012167
8	60.8686	1.54386	1.13794	1.13419	0.0487	0.48457	0.38371	0.14061	1.87131	2.72927	1.25448	0.0011814	0.0011814	0.0011814
9	67.6509	1.56186	1.06303	1.12644	0.92037	0.46164	0.81705	0.41361	1.42095	2.2184	1.11116	0.0011460	0.0011460	0.0011460
10	75.1962	1.54574	1.03515	1.12064	0.995	0.34703	0.85933	0.40179	1.44704	1.90842	1.12248	0.0011106	0.0011106	0.0011106
11	81.9708	1.55617	1.03367	1.12488	0.99337	0.42769	0.85481	0.44941	1.44914	1.6937	1.12669	0.0010753	0.0010753	0.0010753
12	89.4905	1.50249	0.95701	1.0828	0.94437	0.72625	0.83635	0.38593	1.51895	1.46909	1.15051	0.0010399	0.0010399	0.0010399
13	96.2559	1.50545	0.95865	1.09736	0.88283	0.78642	0.8551	0.45102	1.5747	1.27307	1.14502	0.0010045	0.0010045	0.0010045
14	102.875	1.4998	0.93405	1.09727	0.96731	0.82568	0.88067	0.48487	1.44515	1.00086	1.1546	0.0009692	0.0009692	0.0009692
15	110.695	1.48004	0.9152	1.08738	0.93504	0.78649	0.89276	0.47972	1.393	0.95568	1.09595	0.0009338	0.0009338	0.0009338
16	117.431	1.49272	0.91648	1.09863	0.92497	0.77376	0.86334	0.47175	1.39362	0.98169	1.11489	0.0008984	0.0008984	0.0008984
17	125.157	1.5083	0.9214	1.0978	0.93448	0.83391	0.8923	0.48141	1.43973	0.8009	1.15772	0.0008631	0.0008631	0.0008631
18	131.811	1.48133	0.90507	1.09122	0.93026	0.85561	0.89828	0.4684	1.43877	0.80913	1.13495	0.0008277	0.0008277	0.0008277
19	139.302	1.48427	0.88094	1.08528	0.94712	0.85255	0.90272	0.51205	1.31413	0.73582	1.09852	0.0007923	0.0007923	0.0007923
20	145.942	1.4767	0.88706	1.08043	0.97436	0.8751	0.90417	0.52057	1.31846	0.69534	1.09198	0.0007570	0.0007570	0.0007570
21	153.599	1.46925	0.85644	1.07857	0.92639	0.84996	0.90408	0.50907	1.334	0.73224	1.08208	0.0007216	0.0007216	0.0007216
22	160.104	1.46884	0.85446	1.06891	0.93601	0.86691	0.90425	0.53725	1.2945	0.69938	1.07847	0.0006862	0.0006862	0.0006862
23	167.582	1.45631	0.83268	1.08043	0.94443	0.81596	0.90055	0.52791	1.31667	0.68714	1.09877	0.0006509	0.0006509	0.0006509
24	173.905	1.43069	0.83878	1.07589	0.9711	0.85372	0.90677	0.49949	1.39304	0.69246	1.12326	0.0006155	0.0006155	0.0006155
25	181.025	1.39879	0.81153	1.06119	0.98555	0.81579	0.88949	0.52722	1.31569	0.68202	1.08636	0.0005801	0.0005801	0.0005801
26	187.46	1.41548	0.81743	1.06175	0.91969	0.86257	0.9009	0.52894	1.38189	0.70599	1.13772	0.0005448	0.0005448	0.0005448
27	194.433	1.4218	0.79651	1.06274	0.93625	0.88696	0.92281	0.5455	1.26815	0.64721	1.08287	0.0005094	0.0005094	0.0005094
28	200.854	1.37017	0.78152	1.05037	0.93889	0.89357	0.92133	0.53686	1.32008	0.63171	1.09189	0.0004740	0.0004740	0.0004740
29	207.178	1.40284	0.79109	1.05902	0.921	0.88345	0.91967	0.52646	1.35982	0.65585	1.0906	0.0004387	0.0004387	0.0004387
30	214.605	1.38649	0.77851	1.05209	0.95386	0.86684	0.91553	0.539	1.31435	0.63389	1.09254	0.0004033	0.0004033	0.0004033
31	222.038	1.38005	0.76095	1.04054	0.93859	0.85728	0.91655	0.52489	1.34841	0.63906	1.08652	0.0003679	0.0003679	0.0003679
32	229.217	1.35808	0.79932	1.07021	0.96393	0.8575	0.91807	0.52026	1.35362	0.68588	1.06849	0.0003326	0.0003326	0.0003326
33	235.574	1.33463	0.76464	1.07064	0.97761	0.86201	0.91058	0.52775	1.35524	0.65017	1.09417	0.0002972	0.0002972	0.0002972
34	242.439	1.34111	0.76158	1.07462	0.9537	0.86355	0.90904	0.50247	1.48733	0.65585	1.13284	0.0002618	0.0002618	0.0002618
35	248.734	1.34202	0.73707	1.06208	0.96199	0.86532	0.91227	0.52576	1.38574	0.63156	1.09821	0.0002264	0.0002264	0.0002264
36	255.623	1.29683	0.71809	1.05051	0.95196	0.86593	0.91627	0.56025	1.25194	0.59841	1.05678	0.0001911	0.0001911	0.0001911
37	261.704	1.31178	0.72665	1.04771	0.95026	0.89436	0.91714	0.52308	1.45234	0.66641	1.14076	0.0001557	0.0001557	0.0001557
38	268.19	1.31324	0.72455	1.05864	0.95032	0.87039	0.91579	0.5402	1.28825	0.60528	1.06265	0.0001203	0.0001203	0.0001203
39	275.139	1.3389	0.74106	1.0636	0.94453	0.87127	0.91759	0.54745	1.33185	0.61306	1.08521	8.50255e-05	8.50255e-05	8.50255e-05
40	281.147	1.28952	0.71116	1.04515	0.94087	0.87141	0.91905	0.54507	1.31909	0.61067	1.07876	4.96578e-05	4.96578e-05	4.96578e-05

Tablo-2: Eğitim metrikleri

Ek-2: Çalışılan Bilgisayarların Spesifikasyonları

- Donanım Modeli: Lenovo
- İşlemci: Intel® Xeon® W-2235 @ 3.80GHz (12 çekirdek)
- Bellek (RAM): 32.0 GiB
- Grafik Kartı: NVIDIA RTX A4000 (GA104GL)
- Disk Kapasitesi: 512.1 GB SSD
- İşletim Sistemi: Ubuntu 22.04.5 LTS (64-bit)
- GNOME Sürümü: 42.9
- Pencereleme Sistemi: X11